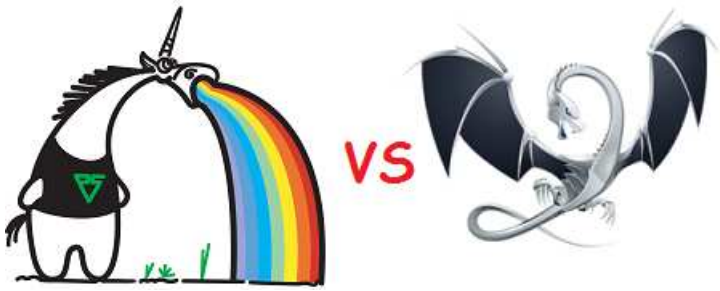
PVS-Studio vs Clang

Author: Andrey Karpov

Date: 05.08.2011

By chance, we have checked the Clang project. I think some developers will be curious about the results.



Presently PVS-Studio uses the external preprocessor of Microsoft Visual C++, which is a large disadvantage. Visual C++'s preprocessor is extremely slow and has some errors we cannot fix. Yes, do not be surprised that the preprocessor works very poor yet it doesn't prevent us from compiling our files in Visual Studio quickly and correctly. I don't know how in particular cl.exe is organized but I can suggest by indirect hints that there are two absolutely different preprocessing algorithms used for compilation and creation of *.i files. Perhaps such an organization is convenient from the viewpoint of compiler's architecture.

We started to actively search for an alternative solution for file preprocessing recently. And it seems that we will choose the preprocessor implemented in Clang. Preliminary testing shows that it works several times faster than cl.exe, which is quite satisfying and useful to us.

Clang is the new compiler for C-like languages (C, C++, Objective-C, Objective-C++, support of C++11 is expected to be implemented soon). Development is sponsored by the Apple Corporation. One of the strong points of the Clang compiler is a large set of static code analysis rules. Actually Clang is already used by Apple as a static analyzer.

Clang was initially designed to save as much information during the compilation process as possible [1]. This feature allows Clang to create detailed context-oriented error reports which are transparent both to developers and development environments. The compiler's module design allows to use it as part of a development environment for the purpose of syntax highlighting and refactoring.

So, perhaps it was the better option to build our PVS-Studio around Clang and not VivaCore [2]. But it's too late now and it's not that simple: in this case we would depend too much on the capabilities of a third-party library. Moreover, Clang's developers take their time supporting Microsoft Specific.

But we've got distracted. It is most interesting to check one code analyzer with another. And so we did it as we had already started studying the Clang project anyway.

Unfortunately, complete comparison is impossible. It would be wonderful to check Clang with PVS-Studio and vice versa and then calculate the number of detected errors for one thousand lines. The trouble is that we are able to check Clang but Clang isn't able to check us. Support of MSVC in Clang is experimental. The matter is also complicated by the fact that PVS-Studio already uses capabilities of

C++11. A mere attempt to compile the project causes error reports saying that 'These language extensions are not supported yet'.

So we will have to content ourselves with what PVS-Studio has managed to find in Clang's code. Of course, we will not describe in this article all of the errors that were detected. Clang's source code is about 50 Mbytes. I will inform its developers about the defects we have found and if they are interested, they can check their project themselves and study the whole list of potential errors. However, they heard of us and discussed some diagnostics of our tool.

Let's see what interesting things are there in the project. Virtually all the detected errors are Copy-Paste errors.

Copy-Paste error N1

```
static SDValue PerformSELECTCombine(...)
{
  ...
  SDValue LHS = N->getOperand(1);
  SDValue RHS = N->getOperand(2);


  ...
  if (!UnsafeFPMath &&
      !DAG.isKnownNeverZero(LHS) && !DAG.isKnownNeverZero(RHS))
  ...
  if (!UnsafeFPMath &&
      !DAG.isKnownNeverZero(LHS) && !DAG.isKnownNeverZero(LHS))
  ...
}
```

PVS-Studio's corresponding diagnostic: V501 There are identical sub-expressions '!DAG.isKnownNeverZero (LHS)' to the left and to the right of the '&&' operator. LLVMX86CodeGen x86isellowering.cpp 11635

In the beginning the code handles both LHS and RHS, but later it handles only LHS. The reason is perhaps a misprint or a copied fragment of a string. As far as I understand, the RHS variable must also be present in the second case.

Copy-Paste error N2

```
MapTy PerPtrTopDown;
```

```
MapTy PerPtrBottomUp;


void clearBottomUpPointers() {
  PerPtrTopDown.clear();
}


void clearTopDownPointers() {
  PerPtrTopDown.clear();
}
```

PVS-Studio's corresponding diagnostic: V524 It is odd that the body of 'clearTopDownPointers' function is fully equivalent to the body of 'clearBottomUpPointers' function (ObjCARC.cpp, line 1318). LLVMScalarOpts objcarc.cpp 1322

This is a classic Copy-Paste. The function was copied; its name was changed but its body wasn't. The correct code is this one:

```
void clearBottomUpPointers() {
  PerPtrBottomUp.clear();
}
```

Copy-Paste error N3

```
static Value *SimplifyICmpInst(...) {
  ...
  case Instruction::Shl: {
    bool NUW = LBO->hasNoUnsignedWrap() && LBO->hasNoUnsignedWrap();
    bool NSW = LBO->hasNoSignedWrap() && RBO->hasNoSignedWrap();
  ...
}
```

PVS-Studio's corresponding diagnostic: V501 There are identical sub-expressions 'LBO->hasNoUnsignedWrap ()' to the left and to the right of the '&&' operator. LLVMAnalysis instructionsimplify.cpp 1891

The developers most likely intended to write this code:

```
bool NUW = LBO->hasNoUnsignedWrap() && RBO->hasNoUnsignedWrap();
```

Copy-Paste error N4

```
Sema::DeduceTemplateArguments(...)
{
  ...
  if ((P->isPointerType() && A->isPointerType()) ||
      (P->isMemberPointerType() && P->isMemberPointerType()))
  ...
}
```

PVS-Studio's corresponding diagnostic. V501 There are identical sub-expressions 'P->isMemberPointerType ()' to the left and to the right of the '&&' operator. clangSema sematemplatededuction.cpp 3240

This is a simple case unlike the fifth sample. It's clear that this code was meant:

(P->isMemberPointerType() && A->isMemberPointerType())

Copy-Paste error N5

```
static bool CollectBSwapParts(...) {
  ...
  // 2) The input and ultimate destinations must line up:
  // if byte 3 of an i32 is demanded, it needs to go into byte 0
  // of the result. This means that the byte needs to be shifted
  // until it lands in the right byte bucket. The shift amount
  // depends on the position: if the byte is coming from the
  // high part of the value (e.g. byte 3) then it must be shifted
  // right. If from the low part, it must be shifted left.
  unsigned DestByteNo = InputByteNo + OverallLeftShift;
  if (InputByteNo < ByteValues.size()/2) {
    if (ByteValues.size()-1-DestByteNo != InputByteNo)
      return true;
  } else {
```

```
    if (ByteValues.size()-1-DestByteNo != InputByteNo)
      return true;
  }
  ...
}
```

PVS-Studio's corresponding diagnostic: V523 The 'then' statement is equivalent to the 'else' statement. LLVMInstCombine instcombineandorxor.cpp 1387

And this is an example of a difficult case. I'm not even sure if there is an error here. The comment also doesn't help me. This code must be studied by its creators. But still I think that this is an example of a Copy-Paste error.

I think that's enough about Copy-Paste for now because there are some errors of other types as well.

Here you are, for instance, a classic error in switch

```
void llvm::EmitAnyX86InstComments(...) {
  ...
  case X86::VPERMILPSri:
    DecodeVPERMILPSMask(4, MI->getOperand(2).getImm(),
                        ShuffleMask);
    Src1Name = getRegName(MI->getOperand(0).getReg());
  case X86::VPERMILPSYri:
    DecodeVPERMILPSMask(8, MI->getOperand(2).getImm(),
                        ShuffleMask);
    Src1Name = getRegName(MI->getOperand(0).getReg());
    break;
  ...
}
```

PVS-Studio's corresponding diagnostic: V519 The 'Src1Name' variable is assigned values twice successively. Perhaps this is a mistake. Check lines: 211, 215. LLVMX86AsmPrinter x86instcomments.cpp 215

Such errors are merely inevitable in a large sized code - so dangerous is the switch operator. No matter how well you know how it works, you can still forget this damned 'break'.

There are some obviously senseless or suspicious operations, if not errors.

Odd operation N1 that can be an error

```
AsmToken AsmLexer::LexLineComment() {
  // FIXME: This is broken if we happen to a comment at
  // the end of a file, which was .included, and which
  // doesn't end with a newline.
  int CurChar = getNextChar();
  while (CurChar != '\n' && CurChar != '\n' && CurChar != EOF)
    CurChar = getNextChar();
  ...
}
```

PVS-Studio's corresponding diagnostic: V501 There are identical sub-expressions to the left and to the right of the '&&' operator: CurChar != '\n' && CurChar != '\n' LLVMMCParser asmlexer.cpp 149

Most likely, the second check CurChar != '\n' is unnecessary here. But perhaps it's an error and then t the following code should be present:

```
while (CurChar != '\n' && CurChar != '\r' && CurChar != EOF)
```

Odd operation N2 that is not an error for sure

```
std::string ASTContext::getObjCEncodingForBlock(...) const {
  ...
  ParmOffset = PtrSize;
  // Argument types.
  ParmOffset = PtrSize;
  ...
}
```

PVS-Studio's corresponding diagnostic: V519 The 'ParmOffset' variable is assigned values twice successively. Perhaps this is a mistake. Check lines: 3953, 3956. clangAST astcontext.cpp 3956

Odd operation N3 I find difficult to describe

```
static unsigned getTypeOfMaskedICmp(...)

{

  ...

  result |= (icmp_eq ? (FoldMskICmp_Mask_AllZeroes |

                        FoldMskICmp_Mask_AllZeroes |

                        FoldMskICmp_AMask_Mixed |

                        FoldMskICmp_BMask_Mixed)

                      : (FoldMskICmp_Mask_NotAllZeroes |

                        FoldMskICmp_Mask_NotAllZeroes |

                        FoldMskICmp_AMask_NotMixed |

                        FoldMskICmp_BMask_NotMixed));

  ...

}
```

PVS-Studio's corresponding diagnostic:

V501 There are identical sub-expressions 'FoldMskICmp_Mask_AllZeroes' to the left and to the right of the '|' operator. LLVMInstCombine instcombineandorxor.cpp 505

V501 There are identical sub-expressions 'FoldMskICmp_Mask_NotAllZeroes' to the left and to the right of the '|' operator. LLVMInstCombine instcombineandorxor.cpp 509

I don't know if these are simple duplicates or another condition must be present. I find it difficult to suggest what exactly should be here.

There is code which is just potentially dangerous.

This code will work as long as two enums have a similar structure.

```
enum LegalizeAction {

  Legal,

  Promote,

  Expand,

  Custom

};
```

```
enum LegalizeTypeAction {
  TypeLegal,
  TypePromoteInteger,
  TypeExpandInteger,
  ...
};

LegalizeTypeAction getTypeAction(...) const;

EVT getTypeToExpandTo(LLVMContext &Context, EVT VT) const {
  ...
  switch (getTypeAction(Context, VT)) {
  case Legal:
    return VT;
  case Expand:
  ...
}
```

PVS-Studio's corresponding diagnostic:

V556 The values of different enum types are compared: switch(ENUM_TYPE_A) { case ENUM_TYPE_B: ... }. LLVMAsmPrinter targetlowering.h 268

V556 The values of different enum types are compared: switch(ENUM_TYPE_A) { case ENUM_TYPE_B: ... }. LLVMAsmPrinter targetlowering.h 270

The name TypeLegal sounds similar to Legal while name TypeExpandInteger is similar to Expand. This has become the cause of the misprint. It's sheer luck that values of these names coincide and therefore the code works.

Conclusion

It's scary when you find errors in a compiler, isn't it? :)

P. S.

It seems to me that my praise of Clang was hasty. We have just come across a situation when it breaks the source code during preprocessing. There is the following fragment in atlcore.h:

```
ATLASSUME(p != NULL); // Too .... here

if (*p == '\0')

  return const_cast<_CharType*>(p+1);
```

Clang's preprocessor turns it into:

```
do { ((void)0);

#pragma warning(push)

#pragma warning(disable : 4548)

 do {__noop(!!(p != 0));} while((0,0)

#pragma warning(pop)

 ); } while(0); // Too .... here if (*p == '\0')

  return const_cast<_CharType*>(p+1);
```

It has put the 'if' operator after the comment and it appears that "if (*p == '\0')" is now a comment too. As a result, we have an incorrect code. Oh, poor us, programmers.

References.

1. Wikipedia. Clang. http://www.viva64.com/go.php?url=714
2. VivaCore Library. http://www.viva64.com/en/vivacore-library/